Cryptocurrencies (WS2024/25) Dr. Zeta Avarikioti, Prof. Matteo Maffei

Kerma - Task 1

1 General Notes

In your group of up to three students, you may use any programming language of your choice to implement the following task. We provide skeleton projects in Python and TypeScript (see `python-skeleton-for-task-1.tgz` and `typescript-skeleton-for-task-1.tgz` on TUWEL). You can use them to build upon. After the deadline, these skeleton projects will be updated and act as sample solutions for the previous tasks. These are guaranteed to pass all test cases for the previous tasks.

High level Overview

After completing this task, your node should be able to:

- listen for incoming connections on port 18018.
- perform the handshake.
- discover and connect to new peers.

Description

Start coding the implementation of your `Kerma`* node. First, start listening at port 18018 and exchange a `hello` message with any peer connecting to you. Then, you will extend your `Kerma` node so that it can exchange messages and perform peer discovery.

- Decide what programming language you will use (we recommend using either Python or Typescript, as these are the languages we provide help with and we release sample solutions).
- Find a cool name for your node.
- Implement the networking core of your node. Your submitted node must listen to TCP port 18018 for connections and must also be able to initiate TCP connections with other peers. Your node must be able to support connections to multiple nodes at the same time.
- Implement canonical JSON encoding for messages as per the format specified in the protocol.

*Out of curiosity, Kerma means “coin" in Greek.

- Implement message parsing, defragmentation, and canonical JSON encoding and decoding. On receiving data from a connected node, decode and parse it as a JSON string. If the received message is not a valid JSON or doesn't parse into one of the valid message types, send an `"error"` message with error name `INVALID_FORMAT` to the node. Note that a single message may get split across different packets, e.g. you may receive `"type":"ge` and `tpeers"` in separate messages. So you should defragment such JSON strings. Alternatively, a single packet could contain multiple messages (separated by `"\n"`) and your node should be able to separate them. Note that JSON strings you receive may not be in canonical form, but they are valid messages nevertheless.
- Implement the protocol handshake:
  - When you connect to a node or another node connects to you, send a `"hello"` message with the specified format.
  - If a connected node sends any other message prior to the hello message, you must send an `"error"` message with error name `INVALID_HANDSHAKE` to the node and then close the connection with that node. Note: Every message you send on the network must have a newline, i.e. `"\n"` at the end. Your node should use this to help parse and defragment valid messages. If you do not receive a `hello` message after 20s or receive a second `hello` message, you should send an error message (again with name `INVALID_HANDSHAKE`) and close the connection.
- Implement peer discovery bootstrapping by hard-coding some peers (for now, only hard-code the bootstrap node `128.130.122.101:18018`).
- Store a list of discovered peers locally. This list should survive reboots.
- Upon connection with any peer (initiated either by your node or the peer), send a `"getpeers"` message immediately after the `"hello"` message.
- On receiving a `"peers"` message from a connected peer, update your list of discovered peers.
- On receiving a `"getpeers"` message from a connected peer, send a `"peers"` message with your list of peers. Note that by specification of the protocol, a `"peers"` message must not contain more than 30 peers. If you have more than 30 peer stored, devise a policy on which 30 you send over the network.
- Devise a policy to decide which peers to connect to and how many to connect to. We suggest to connect to just a few nodes, and not all of them.
- If your node receives a valid message with different type than `hello`, `getpeers` or `peers`, you are not required to determine its validity in this task. You should not, however, close the connection if such a message is sent to you. For instance, if your node receives a `{"type":"getchaintip"}` message, then you should just ignore this message for now.
- Submit your implementation on TUWEL.

Important: Make sure that there are no bugs that crash your node and your node can run for a long time. If our automatic grading script can not connect to your node, you will not receive any credit. Taking enough time to test your node will help you ensure this. Below is a (non-exhaustive) list of test cases that your node will be required to pass. We will also use these test cases to grade your submission.

- *The grader node "Grader" should be able to connect to your node. If you don't pass this test, Grader would not be able to grade the rest of the test cases. So make sure that you test this before you submit.*
- *Grader should receive a valid `hello` message on connecting.*
- *The hello message should be followed by a `getpeers` message.*
- *Grader should be able to disconnect, then connect to your node again.*
- *If Grader sends a `getpeers` message, it must receive a valid peers message.*
- *If Grader sends `{"type":ge`, waits for 0.1 second, then sends `tpeers"}`, your node should reply with a valid peers message.*
- *If Grader sends any message before sending `hello`, your node should send an error message with name set to `INVALID_HANDSHAKE` and then disconnect.*
- *If Grader sends an invalid message, your node should send an error message with the correct name. Some examples of invalid messages are:*
  1. `Wbgygvf7rgtyv7tfbgy{{{`
  2. `"type":"diufygeuybhv"`
  3. `"type":"hello"`
  4. `"type":"hello","version":"jd3.x"`
  5. `"type":"hello","version":"5.8.2"`
- *If grader sends a set of peers in a valid `peers` message, disconnects, reconnects and sends a `getpeers` message, it must receive a `peers` message containing (at least) the peers sent in the first message.*
- *Grader should be able to create two connections to your node simultaneously.*

Important notice - protocol changes from v1 to v2

The section on peer messages has been updated in the protocol description in order to clarify disambiguations. Please re-read the corresponding sections in the protocol description. During grading of the first task, however, the grader will only send syntactically valid peer addresses. You can assume that they run the Kerma protocol; concretely, that they run the same implementation as the bootstrap node. Therefore, if you already implemented this part, you can keep it for now without experiencing disadvantages during grading of the first task.

How to participate in the Kerma network

We suggest you to host your implementation yourself and let it actively participate in the Kerma network. There are multiple options to host your node:

- Use a static[†] and public[‡] IPv4 address. One way to get such an address is make use of free student offers available through (for instance) GitHub Student[§], and to host your node on a VM in the cloud. Upload the static ip address to TUWEL.
- Host your node behind a dynamic[¶], public ip address and provide us with a domain entry that always points to this address.

How to Submit your Solutions

We will grade your solution locally. Please upload your submission as a tar archive. When grading your solution, we will perform the following steps:

```
tar -xf <your submission file> -C <grading directory>
cd <grading directory>
docker-compose build
docker-compose up -d
# grader connects to localhost port 18018 and runs test cases
docker-compose down -v
```

When started in this way, there should be neither blocks (except the genesis block) nor transactions in the node's storage.
If you use the skeleton templates, you can use the provided Makefile targets to build and check your solution. Note however, that this will only check if the container can be built, started and a connection can be established to localhost:18018. We highly recommend that you write your own test cases.
The deadline for this task is 22th October, 11.59pm. We will not accept any submissions after that. Each group member has to submit the solution individually. Plagiarism is unacceptable. If you are caught handing in someone else's code, you will receive zero points.

[†]static means that it will not change over the duration of the course
[‡]public means that it must be accessible from everywhere
[§]See https://education.github.com/pack for more information
[¶]dynamic means that it may change over the duration of the course